EXHIBIT 4

Hashing functions

G. D. Knott
Division of Computer Research and Technology, National Institutes of Health, Department of Health, Education, and Welfare, Bethesda, Maryland 20014, USA

The object of this paper is to survey various hashing functions, to present a brief history of hashing schemes and their development, and to give an exhaustive bibliography on hashing and hash table storage and retrieval methods
(Received January 1972)

1. Introduction

The storage and retrieval problem arises frequently in programming. Often we have given a collection of items presented in arbitrary order, and we wish to store these items and upon demand retrieve those items whose key-values match given key-values. Moreover we may wish to delete previously-stored items and include further new items from time to time.

One particular approach to the storage and retrieval problem is the hash table method of storage and retrieval. The spirit of such schemes is to use the key value of an item to compute an address for the storage or retrieval of that item, Key-values for which the same address is computed are called *synonyms*. The possibility of *collisions* due to the occurrence of synonymous key-values is a major difficulty which can be overcome in various ways.

Because the address calculation is generally a randomising scrambling of the given key-value, the term *hashing** has become the name of this computation. The storage area used to store items is known as a *hash table* and the various algorithms for accessing hash tables and resolving collisions are known as hash table storage and retrieval algorithms (or just hash storage algorithms).

There are two major formulations of hash table storage and retrieval algorithms, differing in the manner in which collisions are resolved. One of these approaches is to establish a hash table for the storage of items and to resolve collisions by somehow finding an unoccupied space for those items whose natural home location (according to the hashing function being used) is already full, in such a way that the item can be later retrieved. Algorithms which use such schemes are called *open addressing* algorithms. The other approach finesses the problem of collisions by using indirect addressing (i.e. simple list-processing techniques) to allow all items which collide to maintain a claim to their home location. These methods of handling collisions are commonly called *chaining* methods.

Often we think of an address as the address of a space which can hold many items rather than just one. Such a storage space is called a *bucket*. Even when the cells in the hash table can hold only a single item, we shall allow all synonymous items to maintain a logical claim to their common initially-addressed bucket, and we shall say that such synonymous items reside in the logical bucket associated with the initially-addressed cell.

In discussing any storage and retrieval method, many complicating factors exist such as the possibility of variable length items, retrieval on secondary keys, and the possibility of duplicate key-values in distinct items. We shall ignore these problems in most of the discussion below. It is often easy to see how to accommodate special circumstances within the framework of a particular basic algorithm. Finally we shall generally confuse an item with its key-value for notational clarity. Thus we speak of an item x with key-value x.

This paper surveys various hashing functions. We then present a brief history of hashing schemes and their development. Finally, an exhaustive bibliography on hashing and hash table storage and retrieval methods is given.

2. Hashing functions

In this section we shall discuss various hashing methods. A hashing function is defined on a domain of values which includes the possible key-values of the items to be processed. The range of a hashing function is some given segment of integers, say $0, 1, \ldots, n - 1$.

For all hash table storage and retrieval schemes characterized by the collision resolution methods mentioned before, a good hashing function is one which minimises the expected number of collisions.

2.1 *Distribution-dependent hashing functions*

Consider a collection, G, of k items with integer key-values which are included in a universe of possible integers, U. (Any set of key-values can be assumed to be integers under an appropriate interpretation). The key-values in G are distributed according to some discrete empirical distribution function, F_X, where F_X is the distribution of the random variable X defined on G as $X(x) = x$; we wish to find a hashing function H with a domain including U and range equal to $\{0, 1, \ldots, n - 1\}$, such that the random variable $H(X)$ defined on G has a discrete uniform distribution function on $\{0, \ldots, n - 1\}$.

In elementary terms we wish to find a hashing function H which maps our k given items to the addresses $0, 1, \ldots, n - 1$ such that any given address is not burdened with more than its share of items; that is, so that the items are uniformly distributed over the address values. The reason for desiring a uniform distribution of the hash-values is to minimise the number of collisions which will occur.

Recall that $F_X(t) = P(X \leqslant t)$, where $P(C)$ denotes the probability of C. We wish to find H such that $P(H(X) = i) = \frac{1}{n}$ for $0 \leqslant i \leqslant n$. Consider the random variable $F_X(X)$. Note that $P\left(F_X(X) \leqslant \frac{r}{k}\right) = \frac{r}{k}$ for $0 < r \leqslant k$. (This is strictly true only when there are no items in G with duplicate key-values. We shall assume that this is the case in this discussion). Thus $F_X(X)$ has a discrete uniform distribution on

$$\left\{\frac{1}{k}, \ldots, \frac{k-1}{k}, 1\right\},$$

and so $nF_X(X)$ has a discrete uniform distribution on $\left\{\frac{n}{k}, \frac{2n}{k}, \ldots, n\right\}$. Therefore $\lceil nF_X(X)\rceil - 1$ is approximately uniform on $\{0, 1, \ldots, n - 1\}$ (especially when $n \leqslant k$).

From the above we see that a good choice for H is $H(x) = \lceil nF_X(x)\rceil - 1$.

When F_X is known and is easily computable, it can be used as above to obtain a hashing function which is tailored to the collection of items to be stored. When F_X is itself a uniform distribution on $\{1, 2, \ldots, h\}$, our hashing function becomes $H(x) = \left\lceil \frac{nx}{h} \right\rceil - 1$ where h is the largest key-value occurring in

*Hashing means to cut or chop; hash is a random jumble achieved by hashing; thus the term is appropriate.

G. One may take h as the largest key-value in U if the largest key-value occurring among the actual items is not known. This particular address calculation function has often been used. An extension to the case where F_X can be approximated by a piece-wise linear function is discussed by Isaac and Singleton (1956) and later by Kronmal and Tarter (1965). Polynominal approximations have been tried by Simon and Guiho (1972).

Tarter and Kronmal have also considered how to make use of the partial knowledge of F_X which is gained by knowing just some of the items being stored (Tarter and Kronmal, 1968). Their scheme is very costly computationally, but it has interest as an example, independent of its utility.

Consider a sample of key-values $x_1, x_2, \ldots, x_m$ taken without replacement from the file of items, G, being stored. Let all the items in G have key-values in the range $[a, b]$. Then the Tarter-Kronmal hashing function, H, which maps key-values in $[a, b]$ to $\{0, 1, \ldots, n-1\}$, is given as

$$H(x) = \lceil n\hat{F}_{mt}(x)\rceil - 1 \ ,$$

where:

$$\hat{F}_{mt}(x) = \tfrac{1}{2} + \frac{x-\bar{x}}{2(b-a)} + \frac{b-a}{2\pi}\sum_{1\leqslant i\leqslant t}\frac{1}{i}\left\{c_i \sin\left[i\pi\frac{x-a}{b-a}\right] - s_i\cos\left[i\pi\frac{x-a}{b-a}\right]\right\}$$

and where,

$$\bar{x} = \frac{1}{m}\sum_{1\leqslant i\leqslant n} x_i \ ,$$

$$c_i = \frac{2}{(b-a)n}\sum_{1\leqslant j\leqslant m}\cos\left[i\pi(x_j-a)/(b-a)\right] \ ,$$

$$s_i = \frac{2}{(b-a)n}\sum_{1\leqslant j\leqslant m}\sin\left[i\pi(x_j-a)/(b-a)\right] \ ,$$

and t = the least integer $\geqslant 1$ such that

$$\tfrac{1}{4}(b-a)^2\,(c_{t+1}^2 + s_{t+1}^2) \leqslant 2\left(\frac{n-1}{n-m}\,m + 1\right)^{-1} .$$

$\hat{F}_{mt}$ is an optimised t-term Fourier series approximation of the empirical distribution function F_X for the items in G. Let $F_m^*(x) = P(X \leqslant x \mid X \in \{x_1, \ldots, x_m\})$. Thus F_m^* is the empirical distribution of the sample $\{x_1, \ldots, x_n\}$, and as such is an approximation to F_X. Then $\hat{F}_{mt}(x) = \frac{1}{2}\frac{x-a}{b-a} + F_{mt}^*(x)$, where F_{mt}^* is the t-term Fourier series approximation of

$$F_m^*(x) - \tfrac{1}{2}\frac{x-a}{b-a} .$$

The subtraction of the $\frac{1}{2}\frac{x-a}{b-a}$ term which is subsequently added back in is a device to increase the rate of convergence of the Fourier series for F_m^*.

Tarter and Kronmal have also given a way to update the values $\bar{x}$, c_i, s_i, and t when more items are known, i.e. when m increases. Thus $\hat{F}_{mt}$ can be made to be a better and better approximation to F_X. The use of H as an adaptive function necessitates the use of memory to store the previous states of H in so far as required for retrieval. In this sense, H is related to the methods discussed in Knott (1971).

Another possible scheme to make use of partial knowledge of F_X is to use the key-value x to determine 'where' in the current collection of already-stored items we are, and then to select an appropriate function (possibly just a constant) to use as our hash function. This approach is inspired by Marsaglias' method of generating pseudorandom numbers which are distributed in some given manner; see Knuth (1969). Methods of this form which dynamically change the size of the hash table as required are discussed in Knott (1971).

2.2. *Cluster-separating hashing functions*

A general consideration often mentioned is that a hashing function for a collection of items, G, should destroy *clusters* in G, meaning that if the key-values of two items are close to each other in some sense, then their hashed-values are not equal. But no matter what hashing function we choose, there are notions of what constitutes a cluster for which our hashing function fails to achieve reasonable results. Some desirable behaviour can be determined for a prospective hashing function only if one knows something of the distribution of items in G.

For example, it is wise to ponder the nature of identifier strings actually occurring in programs before choosing a hashing function for a hash table storage and retrieval algorithm which is intended to store such strings.

Also, when handling character string keys one may pay particular attention to the non-uniform nature of the binary codes used for representing characters.

In any event, we see that there is no such thing as a general purpose hashing function.

What we mean, therefore, when we speak of an acceptable general hashing function is that we have a hashing function which achieves a reasonably uniform distribution of hash-values, given certain general assumptions about the distribution of the items to be processed.

One common assumption is that the key-values of our collection, G, of items is a random sample (without replacement) from the set of all possible key-values, U. From this point of view, a hashing function with range $\{0, 1, \ldots, n-1\}$ which partitions the universe of possible key-values, U, as nearly as possible into n equi-numerous sets of synonymous key-values corresponding to the addresses $0, 1, \ldots, n-1$ is preferable to one which partitions U into differently sized sets of synonymous key-values. We shall call such a hashing function a *balanced* hashing function. Ullman (1972) would call a hashing scheme which uses a balanced hashing function 1-uniform.

A more local assumption is that key-values tend to occur in arithmetic progression, i.e. if x is an occurring key-value, $x + s$ for some fixed s is relatively likely to occur also. Depending upon the choice of s, this assumption may be subsumed by the situation discussed below. A simple Markov chain model for clustering which is similar to arithmetic progression clustering is discussed by Toyoda, Tezuka, and Kashara (1966). Let us suppose that a reasonable notion of the distance between two key-values is the so-called Hamming metric (1950). Two key-values, x and y, considered to be t-tuples whose components are symbols of some alphabet, are separated by a distance d under the Hamming metric when x and y have different symbols in exactly d different component positions. Another common clustering assumption is that if x is a key-value actually occurring in some item, then key-values which are close to x with respect to the Hamming metric are relatively likely to actually occur in some item also.

If we consider our key-values to be English words of t or fewer letters for example, then the key-values are distributed in a certain manner in the set of all t-component vectors of letters (including blank). This distribution is such that our intuitive notion of clusters is indeed approximated by using the Hamming metric as a measure of closeness. If w is an English word, then w changed in one or two positions is also likely to be an English word, relative to the probability that w changed in one or two positions is an English word given only that w is a random string of letters.

When the notion of cluster can be usefully defined in terms of the Hamming metric, there are methods of constructing 'cluster-destroying' hashing functions which may be studied theoretically using the results of algebraic coding theory. This was first suggested by Muroga (1961).

In particular let our key-values be considered to be vectors of t symbols, padded if necessary, where the alphabet of usable symbols is in 1-1 correspondence with elements of some finite field, $GF(q)$, of q elements. The common situations are to consider the key-values as vectors of bits in correspondence with $GF(2)$, or to consider them to be vectors of binary-coded characters (say b-bit characters) which are then in correspondence with $GF(2^b)$.

The 1-1 correspondence of symbols to elements of a finite field induces an addition and multiplication operation on the appropriately extended alphabet of symbols in the obvious way. Thus we shall speak of the sum or product of two symbols and understand that the symbol in correspondence with the sum or product of the corresponding field elements is intended. Henceforth, we shall speak of the symbol field in which the symbols are embedded rather than the symbol alphabet.

Now that the set of symbols can be considered to be in a finite field, $GF(q)$, for some q, we have that the set of all possible key-values may be considered as a t-dimensional vector space over $GF(q)$.

Now let us further insist that our 'space of addresses' to be computed be the set of all s-tuples over the symbol field. Such an s-tuple is of course interpretable as an integer address in base q notation, or directly as an integer using the string of bits obtained by considering the concatenated binary representations of the various elements of $GF(q)$.

Thus, for example, we can consider mapping 64-bit key-values into 16-bit addresses by mapping 8-tuples over $GF(2^8)$ into 2-tuples over $GF(2^8)$. Such a mapping is of course a hashing function.

But mappings of this form occur in coding theory. There, we take a message of s symbols taken from $GF(q)$ and encode that message as a longer message of t symbols over $GF(q)$, where the extra symbols are used to provide redundancy or error checking information such as parity checks. The t-symbol coded message is then transmitted to some receiving station, where the received message, now possibly in error, is decoded to yield what is considered to be the original s-tuple. The decoding process tries to use the redundancy of the received message to detect and possibly correct any transmission errors which may have occurred. Coding theory is concerned with the design of coding and decoding schemes and the study of their error detection and error correction properties.

Here we are considering mappings which arise in decoding schemes but we consider them only as compression functions with certain cluster-destroying properties.

A hashing function, H, based on methods used in decoding (actually in computing the so-called syndrome of a received message) can be defined on our t-component keys considered as column vectors by $H(x) = Tx$ where T is an $s \times t$ matrix with entries in the symbol field.

The following theorem applies:

Theorem:
(Sacks (1958)) Let T be an $s \times t$ matrix over a field F. Let x and y be t-dimensional column vectors over F. Let D be the Hamming metric. Then for $1 \leqslant d \leqslant s$, $0 < D(x, y) \leqslant d$ implies $Tx \neq Ty$, iff every set of d columns of T is linearly independent over F.

Proof:
Suppose $x \neq y$, and $0 < D(x, y) \leqslant d$, and yet that $Tx = Ty$. Then $T(x - y) = 0$. Now $x - y$ has at most d non-zero components, and thus $T(x - y)$ is a linear combination over F of at most d columns of T, namely those which are selected by the non-zero components of $x - y$. But if every d columns of T are linearly independent, then in particular $T(x - y) = 0$ implies $x - y = 0$ and hence $x = y$. Therefore for $x \neq y$ we have $0 < D(x, y) \leqslant d$ implies $Tx \neq Ty$, given that every d columns of T are linearly independent.

Now, conversely we can choose x and y such that $D(x, y) = d$ and such that $T(x - y)$ is any desired linear combination of any specified d columns of T. But then $Tx \neq Ty$ implies $T(x - y) \neq 0$, for any such choices of x and y, and hence no d columns of T are linearly dependent. QED.

Let us call a hashing function, H, *d-separating with respect to the metric D* if $0 < D(x, y) \leqslant d$ implies $H(x) \neq H(y)$. We then have that $H(x) = Tx$, where T is an $s \times t$ matrix as defined above, is a d-separating hashing function with respect to the Hamming metric if every d columns of T are linearly independent.

There are constraints on the values d, t, and s. In particular we have the so-called Varsharmov–Gilbert bound (Peterson 1961); an $s \times t$ matrix T can be found with elements in a symbol field of q elements such that $H(x) = Tx$ is d-separating whenever t, s, and d are such that:

$$q^s - 1 > \sum_{1 \leqslant i \leqslant d-1} \binom{t-1}{i} \cdot (q-1)^i \ .$$

Thus for $q = 2^8$ (8-bit symbols) and $t = 8$ and $s = 2$, we can certainly achieve $d = 2$, whence no two 8-tuples (64 bit key-values) will map to the same 2-tuple (16-bit bucket address), whenever they differ in less than 3 symbol positions.

Note also that for $d = 1$ and $s \geqslant 1$, t can be arbitrarily large. Thus two keys which differ in just one position will always be mapped to different addresses. In particular, for q prime, we can take $H(x)$ to be the sum modulo q of the t symbols in the key value x, where each symbol is an integer in $\{0, 1, \ldots, q - 1\}$. x can thus be considered as a sequence of t consecutive $\lceil \ln_2 q \rceil$-bit fields to be added together modulo q.

To prove the above bound, we consider constructing a matrix, T, of s-component columns with d fixed. Then by the previous theorem we must choose the columns of T such that every subset of d or fewer columns are independent. Suppose we have chosen $t - 1$ columns such that no d of these columns are linearly dependent. Now in the worst case, every non-zero linear combination of no more than $d - 1$ columns already chosen is a distinct vector which, of course, may not be chosen to be column t. Counting the number of different non-zero linear combinations of at most $d - 1$ existing columns, we obtain the sum given above on the right-hand side. Now we will certainly be able to choose another column if there are any vectors among the $q^s - 1$ non-zero vectors which are not a linear combination of at most $d - 1$ existing columns; and this will surely be the case when the inequality above holds. Therefore when the inequality above holds, there is surely an $s \times t$ matrix T which has every subset of d columns linearly independent and this proves the above statement.

There is a better way to define d-separating hashing functions which map t-tuples to s-tuples than actually computing $H(x) = Tx$ for some appropriate matrix T. The basic idea is to use an equivalent formulation which involves polynomial division as opposed to matrix multiplication.

Let $g(z)$ be a polynomial of degree s with coefficients in the symbol field. Also, consider t-tuples $x = (x_0, x_1, \ldots, x_{t-1})$ to be in correspondence with polynomials $x(z)$ of degree $t - 1$ or less, where $x(z) = x_0 + x_1 z + \ldots + x_{t-1} z^{t-1}$. We shall write $x(z)^*$ to denote the associated t-tuple x.

Now we can consider the hashing function

$$H(x) = (x(z) \bmod g(z))^* \ ,$$

where the result is the s-tuple corresponding to the polynomial $x(z) \bmod g(z)$. Of course all arithmetic among coefficients is done in the symbol field. Knuth (1969) gives a standard polynomial division algorithm which may be used to compute $H(x)$.

The mapping $x(z)^* \to (x(z) \bmod g(z))^*$ is a linear transformation on the space of t-tuples over the symbol field into the space of s-tuples over the symbol field. This follows since for a and b in the symbol field, we have:

$$((ax(z) + by(z)) \bmod g(z))^* = a(x(z) \bmod g(z))^* + b(y(z) \bmod g(z))^* .$$

But then there exists an $s \times t$ matrix T_g such that $T_g x = (x(z) \bmod g(z))^*$ for every t-tuple x. Thus for some matrices, T_g, $T_g x$ can be computed as $(x(z) \bmod g(z))^*$ and for every polynomial $g(z)$ of degree s there is an associated linear transformation, T_g, which can be used to specify a hashing function. It is particularly convenient to compute $T_g x$ by computing $(x(z) \bmod g(z))^*$.

Now we may ask what properties of $g(z)$ result in T_g having every d columns independent for d as large as possible.

Let $K(T_g)$ be the kernel of T_g, that is the set of t-tuples, x, such that $T_g x = 0$. Now if when $x, y \in K(T_g)$ and $x \neq y$ then $D(x, y) > d$, we then have that $H(x) = (x(z) \bmod g(z))^* = T_g x$ is d-separating. To see this suppose that $x, y \in K(T_g)$ and $x \neq y$ implies $D(x, y) > d$. Now let u and v be t-tuples such that $u \neq v$ and $H(u) = H(v)$. Then $D(u, v) = D(u - (u(z) \bmod g(z))^*, v - (v(z) \bmod g(z))^*) > d$ since $(u - (u(z) \bmod g(z))^*) \in K(T_g)$ and $(v - (v(z) \bmod g(z))^*) \in K(T_g)$; and hence H is d-separating as required.

Thus if we choose $g(z)$ such that $K(T_g)$ contains no vectors $x, y, x \neq y$, where $D(x, y) \leqslant d$, then $H(x) = (x(z) \bmod g(z))^*$ is d-separating. In coding theory terms, $K(T_g)$ is called the cyclic code generated by $g(z)$ with minimum distance $d + 1$.

It is not hard to find appropriate polynomials $g(z)$ with which we may construct d-separating hashing functions, but some knowledge of the theory of finite fields is required and we shall forgo these constructions here, (the appendix contains further details).

One may also find polynomials which lead to hashing functions which separate key-values which differ in bursts, that is, in component positions which are within a certain distance of each other. Of course there are various restrictions on the values of t, s, and d, also related to circumstances which involve finite field theory.

Schay and Raver (1963) may be consulted to learn how to choose particular polynomials, $g(z)$. Klinkhamer has also provided an introduction to this subject.

2.3. *Distribution-independent hashing functions*

We now consider some classes of hashing functions which may be applied without specific knowledge of the initial distribution of items. From such a position of ignorance, one may turn to 'scrambling' methods inspired by methods of computing pseudo-random numbers. Some of these methods in fact turn out to be cluster-separating schemes also. We may consider a key-value as a bit-string. Then some common elementary hashing functions are listed below.

1. (extraction). $H(x)$ = a k-bit value composed by extracting k bits from x and concatenating them in some order. The range is $\{0, \ldots, 2^k - 1\}$.
2. (multiplication). $H(x)$ = a k-bit value extracted from the value x^2 or cx for some constant c.
3. (exclusive-oring). $H(x)$ = a k-bit value extracted from the result of exclusive-oring various segments of x together.
4. (addition). $H(x)$ = a k-bit value extracted from the result of adding various segments of x together.
5. (division). $H(x) = x \bmod n$. The range is $\{0, \ldots, n - 1\}$.
6. (radix conversion). $H(x)$ = a k-bit value extracted from the result of treating x as a binary-coded sequence of base p digits and converting this coded value into a similarly-coded base q value. The range is nominally $\{0, \ldots, 2^k - 1\}$.

From such operations as those just given and other simple logical and arithmetic operations one can construct reasonable hashing functions. For example, for character-string key-values, a possible hashing function is to take the residue modulo n of the concatenation of the first character, the last character and the binary value representing the length of the given character string.

In general, one can convert long key-values to a size which is more reasonable for computation by the extraction or exclusive-oring of subfields. When a suitably-sized number is obtained by such compression it may be further transformed by multiplication, radix-conversion or division operations.

The commutativity of certain operations may be undesirable. For example, if we choose to exclusive-or two component fields, a and b, of the key-value; and if there is a high probability that if there is an item with a key-value such that $a = u$, $b = v$, then there will be an item with a key-value such that $a = v$, $b = u$, then the exclusive-oring of the component fields a and b, will lead to a relatively high incidence of collisions. This may be the case when we are storing matrix elements, using their indices as key-values. A solution is to use a non-symmetric compression operation. For example, the first component shifted circularly some number of places may be exclusive-or'ed with the unshifted second component.

The multiplication method is sensitive to the distribution of key-values, and/or the choice of a constant multiplier. The occurrence of many 0 digits in the key-value or constant multiplier results in a simple linear combination of shifted forms of the key-value and/or constant multiplier. The digits of the key-value may then be carried over to the hash-value in a non-random manner, thus biasing the resulting hash-values. This is not serious if we choose a good constant c. Floyd (1970) has discovered an interesting multiplicative scheme which is discussed below. The use of a middle segment of x^2 has a long history, inspired no doubt by Von Neumann (see Knuth, 1969), but other methods may be safer.

The exclusive-or operation has the advantageous feature that if the arguments have 0 or 1 bits occurring equi-probably and independently in each position, then the resulting bit string also has this property.

Addition schemes which extract other than the low-order k-bits of the sum are generally inferior to the use of shifting and exclusive-oring methods in that the number of 0's and 1's resulting do not occur equi-probably and independently even though the input had this property; however, sums of k-bit fields taken modulo 2^k are as good as exclusive-oring. In any event, if the sub-fields involved are not independent, the results of addition (or exclusive-oring) will be biased.

Lin (1963) suggests that radix conversion methods with the initial and final radix values being relatively prime is a good general purpose hashing function.

An attractive particular radix conversion hashing function might be to treat a key-value, x, as a binary value and convert it to a binary-coded base n value, thereupon choosing the low-order digit ($\lceil \ln_2 n \rceil$ bits) as $H(x)$, but this is just the same as computing $x \bmod n$. More complex radix conversion is probably not worthwhile.

Simple division hashing functions seem to have acceptable properties. With a suitable choice of modulus, clusters under the Hamming metric are destroyed, and such hashing functions are usually balanced. Toyoda (1966) has considered the behaviour of a simple division hashing function for a set of items whose keys are clustered according to a simple Markov chain generating scheme. His model formalises the notion of arithmetic progression clusters, but his results are not directly useful without further computation. Knuth (1973) has noted that n should not have 2 or 3 as factors. $x \bmod n$ is even whenever x is even for example. The choice of a prime modulus is often recommended, for then runs of key-values in arithmetic progression hash to different address values as much as possible unless the step-size of the progression is a multiple of the modulus. Buchholz (1963), Heising (1963), and Maurer (1968) have discussed the division method. Buchholz points out that primes of the form $kr^t + 1$, where r is the radix of the key-values (generally a power of 2) may not be the best choice, for then the hash-values are complex shifted sums of subfields of the key-values, and if these subfields are not independent then the comment concerning addition hashing functions applies. The modulus need not necessarily be chosen to be prime, however, and this flexibility is one of the assets of the division method.

Another is that the quotient as well as the remainder is generally available, and this may be put to good use.

When simple extraction alone, or some other 1-1 function on a subfield of the key-value, can be used, one may obtain the advantage that that subfield of the key-value of an item need not be stored since it can be computed from the result of the hash function. Such fields can be dropped only when the collision resolution scheme being used permits it by maintaining the distinction between buckets. This space-saving can be very important in some circumstances, even at the cost of retrieval time.

2.4. *A multiplicative hashing function*

Floyd's construction of a particular multiplicative hashing function (1970) is given in this section.

Let x be as usual a non-negative integral key-value. Let c be such that $0 \leqslant c < 1$. Now we may compute $cx \bmod 1$, i.e. the fraction part of cx, to obtain a value in the unit interval $[0, 1)$. Now given such a value $0 \leqslant y < 1$ we may scale y to obtain an address in $\{0, 1, \ldots, n-1\}$ by computing $\lfloor ny \rfloor$. Thus we will consider a hashing function $H(x)$ of the form

$$H(x) = \lfloor n(cx \bmod 1) \rfloor \ .$$

The question to be resolved here is how to choose c.

Let us assume that key-values have a tendency to occur in arithmetic progression, so that we may have key-values x, $x + s$, $x + 2s, \ldots$. In this case it is desirable that $H(x)$, $H(x + s)$, $H(x + 2s), \ldots$ be a sequence of distinct addresses (in so far as this is possible). But this goal is achieved if the set of values $\{cx \bmod 1, c(x + s) \bmod 1, \ldots\}$ are 'well-spread' in the unit interval, for then the addresses achieved by scaling will be well-spread among $\{0, 1, \ldots, n-1\}$.

Now let us further suppose that $c = \dfrac{k}{h}$ where k and h are integers. It is convenient to choose h such that $s \equiv 1 \bmod h$ where the integer s is the arithmetic progression stepsize. Then for integral a and i we have:

$$c(x + as^i) \bmod 1 = \left(cx + \frac{1}{h}(kas^i \bmod h)\right) \bmod 1$$
$$= (cx + ac) \bmod 1 \ .$$

We have used the following facts:

$$f \bmod 1 = \frac{1}{g}(fg \bmod g) \ ,$$

and

$$(f + g) \bmod 1 = (f + (g \bmod 1)) \bmod 1 \ ;$$

as well as the condition $s^i = 1 \bmod h$. Thus for h and s such that $s \equiv 1 \bmod h$ we have, regardless of the values of $i_1, i_2, \ldots$; that the sequence of key-values $x, x + s^{i1}, x + 2s^{i2}, \ldots$ corresponds to the values $cx \bmod 1$, $(cx + c) \bmod 1$, $(cx + 2c) \bmod 1, \ldots$.

It is no loss of generality to assume x is such that $cx \bmod 1 = 0$ since we here consider the unit interval to be circular $(0 = 1)$ in any event. Hence it suffices that for $s \equiv 1 \bmod h$ we further choose h and k, and thus determine c, such that the points 0, $c \bmod 1$, $2c \bmod 1$, $3c \bmod 1, \ldots$ are well-spread in the unit interval.

We shall consider a set of points to be well-spread in the unit interval when the ratio of the minimum of the lengths of the sub-intervals defined by the given points placed in the unit interval to the maximum sub-interval length is large. We denote this ratio by D_c.

The distribution of 0, $c \bmod 1, \ldots$ has been studied asymptotically by Zaremba (1966). We are interested in the distribution in its early stages as well. Halton (1965) has results of use here.

Now consider the equations:

$$\begin{aligned} 1 &= m_1 c + c_2 \\ c &= m_2 c_2 + c_3 \\ c_2 &= m_3 c_3 + c_4 \\ &\vdots \\ c_{i-1} &= m_i c_i + c_{i+1} \ . \\ &\vdots \end{aligned} \qquad (0)$$

Where each m_i is a non-negative integer and $c_1 = c$ and $0 \leqslant c_i < 1$ for $i \geqslant 1$.

To avoid complications let us suppose that c is irrational: then the sequence $c, c_2, c_3, \ldots$ is not eventually periodic and the following analysis holds. Of course the actual value of c which we will use will be a rational approximation, $\dfrac{k}{h}$, to some desired irrational value, such that $s \equiv 1 \bmod h$.

By considering the process of successively placing the points $c \bmod 1$, $2c \bmod 1, \ldots$ in the unit interval, and keeping in mind the equations (0), we see that whenever a new point is placed it splits a largest sub-interval into two smaller subintervals.

In fact at any time there are at most three differently-sized sub-intervals; and at certain points, which we shall call resting points, only two differently-sized sub-intervals. The table below shows the process at the so-called resting points.

interval sizes	*resting point numbers*			
	0	1	2	. . .
1	1			
c	0	m_1		
c_2		1	$m_2 m_1 + 1$	. . .
c_3			m_1	
$\vdots$				
$N + 1$:	1	$m_1 + 1$	$m_2 m_1 + m_1 + 1$	

(N = number of points placed)

After placing 0 points (at resting point 0) we have 1 interval of size 1 and 0 intervals of size c. After placing m_1 points we have m_1 intervals of size c and 1 interval of size c_2. This follows from (0). Now each interval of size c requires m_2 points to be placed to split it into m_2 intervals of size c_2 and one interval of size c_3. Thus placing $m_2 m_1$ more points takes us to resting point 2 with $m_1 + m_1 m_2$ total points placed and $m_2 m_1 + 1$ intervals of size c_2 and m_1 intervals of size c_3. In general after having placed N points to arrive at resting point $i - 1$ with a intervals of size c_{i-1} and b intervals of size c_i, we then arrive at resting point i after placing $m_i a$ more points, and at that point we have partitioned the unit interval into $m_i a + b$ intervals of size c_i and a intervals of size c_{i+1}.

Let the polynomials Q_i of i arguments be defined by

$$Q_0 = 1 \ ,$$
$$Q_1(x_1) = x_1 \ ,$$

and

$$Q_i(x_1, \ldots, x_i) = x_i Q_{i-1}(x_1, \ldots, x_{i-1}) + Q_{i-2}(x_1, \ldots, x_{i-2}) \quad \text{for } i \geqslant 2 \ ,$$

following Knuth (1969), Section 4.5.3. We shall freely use the elements of the theory of continued fractions given there.

We shall write Q_i without arguments to denote $Q_i(m_1, m_2, \ldots, m_i)$.

Now by considering the above table we have that resting point i corresponds to $N = Q_i + Q_{i-1} - 1$, whereat we have Q_i intervals of size c_i and Q_{i-1} intervals of size c_{i+1}.

At all points between resting point i and resting point $i + 1$ we have a minimum sub-interval of size c_{i+2} and a maximum sub-interval of size c_i so that the measure of well-spreadness, D_c, is given by $D_c = c_{i+2}/c_i$ except at resting points.

Now to compute c which maximises D_c except possibly at resting points we follow Halton (1965). Define $\delta_1 = c$ and $\delta_i = c_i/c_{i-1}$ for $i > 1$. Thus $c_i = \delta_1\delta_2, \ldots, \delta_i = \delta_i c_{i-1}$.

Thus from (0) we have:

$$\frac{c_{i-1}}{c_i} = m_i + \frac{c_{i+1}}{c_i}$$

so,

$$\frac{1}{\delta_i} = m_i + \delta_{i+1}$$

or,

$$m_i = \left\lfloor \frac{1}{\delta_i} \right\rfloor \text{ and } \delta_{i+1} = \left(\frac{1}{\delta_i}\right) \bmod 1 \ .$$

Also,

$$D_c = \delta_1\delta_2 \ldots \delta_{i+2}/\delta_1 \ldots \delta_i = \delta_{i+1}\delta_{i+2} \ .$$

Let $\delta = \min(\delta_{i+1}, \delta_{i+2})$. We have

$$1 = \delta_{i+2}\delta_{i+1} + \delta_{i+1}m_{i+1}$$

so $1 \geqslant \delta^2 + \delta m_{i+1}$ and hence $0 \geqslant \delta^2 + \delta m_{i+1} - 1$. Thus δ is no greater than the greatest root of $0 = \delta^2 + \delta m_{i+1} - 1$ so $\delta \leqslant (-m_{i+1} + \sqrt{m_{i+1}^2 + 4})/2 \overset{df}{=} \gamma$. Also $\delta > 0$ and hence $\delta = \gamma$ only if $\delta = \delta_{i+1} = \delta_{i+2}$. Moreover γ is greatest when m_{i+1} is least. Thus D_c is maximised when $\delta_{i+1} = \delta_{i+2}$ and $m_{i+1} = 1$ for all $i \geqslant 0$. Let us take $m_1 = m_2 = \ldots 1$.

Then successive substitution in (0) to eliminate $c_2, c_3, \ldots$ yields

$$c = \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ddots}}}$$

or,

$$c = \frac{1}{1 + c} \text{ , whence } c = \frac{-1 + \sqrt{5}}{2} \approx 0.618034 \ .$$

Note that we have $c = \phi - 1 = \phi \bmod 1 \overset{df}{=} \rho$ where ϕ is the golden ratio (see Knuth, 1968). For $c = \rho$ we have $\delta_i = \rho$ for all $i \geqslant 1$, and thus the conditions which insure D_c is maximal, except possibly at resting points, are met.

From the above we may claim that the sequence $\rho \bmod 1$, $2\rho \bmod 1, \ldots$, is continually well-spread in the unit interval; in fact it is more uniformly distributed than a random sample of the same size from a uniform distribution. Such a 'super-uniform' sample generator may be of some interest independent of its application here.

The way in which $\rho \bmod 1$, $2\rho \bmod 1, \ldots$ splits the unit interval into sub-intervals is amazingly regular. In particular we have $Q_i = F_{i+1}$, the $i + 1$st Fibonacci number; the ith least-sub-interval size $\rho_i = \rho^i$; and at resting point i which occurs at $N = F_{i+2} - 1$, we have F_{i+1} intervals of size ρ^i and F_i intervals of size ρ^{i+1}. As we move from resting point i to resting point $i + 1$, we place F_{i+1} points, each of which splits an interval of size ρ^i into two sub-intervals of sizes ρ^{i+1} and ρ^{i+2}. The ρ^i-intervals are split in order by age, the oldest (earliest-created) ρ^i-interval being split first. The left-most sub-interval of the pair created is the ρ^{i+1}-sub-interval if i is even and the ρ^{i+2}-sub-interval if i is odd. The ρ^i-intervals are not split in order by ordinal position in the unit interval of course (after all we expect our points to be continually well-spread), but the ordering of the ordinal numbers corresponding to the oldest-first ordering of splitting is given by the sequence (S_j), $j = 1 : F_{i+1}$ where

$$S_j = 1 + ((F_{i+1} + S_{j-1} - 1) \bmod F_{i+2}) \quad \text{for } j > 1$$

and

$$S_1 = 1 \text{ if } i \text{ is even and } S_1 = F_{i+2} \text{ if } i \text{ is odd} \ .$$

The sequence (S_j) is a permutation of $1, 2, \ldots, F_{i+1}$.

Recall we have $c = \frac{k}{h}$ where $s \equiv 1 \bmod h$. From the above analysis, we may choose k such that $\frac{k}{h}$ is a good rational approximation to ρ. For example, we may take $s = 256$, having in mind the storage of items with 8-bit character string key-values, and then we may take $h = 255$ and $k = \lfloor h\rho \rfloor = 158$, whence Floyd's hashing function becomes

$$H(x) = \lfloor n(158x/255 \bmod 1) \rfloor \ .$$

Knuth has developed another scheme for computing

$$H(x) = \lfloor n(cx \bmod 1) \rfloor$$

which involves no division (1973). c is chosen as a rational fraction, $\frac{k}{h}$, with $h = 2^m$ where we are dealing with m-bit values in our arithmetic operations. k is an integer prime to h. Now if we choose n to be a power of 2, say 2^p, then $\lfloor n(cx \bmod 1) \rfloor$ is just the high p bits of the low order word of the double length product kx.

When using this scheme the analysis above does not apply. We still wish $c \approx \rho$, but Knuth has shown that slight modifications in the low order bytes of c are desirable.

2.5. *Index function-based hashing functions*

Index functions, or storage mapping functions, are functions which compute the linear index (i.e. relative address) of an array reference, given the associated subscript values. Such functions exist for rectangular arrays, triangular arrays, the bands of band matrices, and other exotic forms of arrays. The most commonly used index function is the rectangular index function:

$$f(i_1, \ldots, i_s) = \sum_{1 \leqslant j \leqslant s} (i_j - b_j) \cdot \prod_{j+1 \leqslant k \leqslant s} (h_k - b_k + 1)$$

where

$$b_j \leqslant i_j \leqslant h_j \text{ for } 1 \leqslant j \leqslant s \ .$$

An index function which computes a relative location for the cell indexed by $(i_1, \ldots, i_s)$, is of course a hashing function defined on the set of possible vectors of indices. The storage and retrieval problem is trivial since no collisions occur.

We can, however, consider using index functions applied to various fields $i_1, \ldots, i_s$ of a key-value or hashed key-value to obtain a result which may perhaps be further compressed to yield a final hash value. Generally such a hashing function has no more merit than other simpler schemes.

There is, however, a feature of index functions which can occasionally be used to advantage; namely an index function is a 1-1 function onto the integers $0, 1, \ldots, m$ for some m. The number of bits required to express an index function value, v, for an index vector $(i_1, \ldots, i_s)$ is the minimal number of bits which still allows v to be inverted to obtain $(i_1, \ldots, i_s)$. An index function is thus a compression function of its arguments with the advantage that no collisions (identical values) will occur as a result of compressing non-identical arguments. Hence even if the index function values must now be hashed to further compress them, so that collisions become inevitable, the number of collisions may be smaller than if we started by using a reasonable hashing function applied directly to the original key-values, $(i_1, \ldots, i_s)$, which presumably are scattered randomly in a larger 'space' than are the associated index function values.

Moreover if items consist of independently-coded fields of data then computing and storing the value of an index function applied to these fields can save considerable space in the storage area required. If in addition the key-values to be used are virtually the entire item then a simple extraction hashing function of an initially computed index function value is often feasible and in this case, as we have noted above, even further space may be saved. Hence when extreme data compression is required in a hash table storage and retrieval scheme, the use of an index function should be considered.

If the hashing function value is just an extracted field from an appropriate place in an initially-computed index function value, then one can often retrieve items specified by giving values for some of the initial key-value component fields (indices) and considering the other indices to have any arbitrary values. Such handling of 'don't care' conditions is possible since such a retrieval request corresponds to a set, A, of integer key-values, which are the associated index function values for the sought-after items. When the set A is just a small number of intervals, and when the final hash-values can easily be tested to see if their index function value predecessors lie in certain intervals (which often holds for extraction hashing functions) then certain such 'and-form retrievals with don't-care conditions' can be handled. Lum (1970) has elaborated this scheme by proposing that index-function hash index tables be constructed for each of several permutations of the key-value component fields. When a retrieval request occurs, whichever established permutation index-function that is convenient is used. The intent is to use that one which gives us the smallest set of intervals to be used as indices in our search.

We shall now describe another index function-based hashing function scheme which may apply in other circumstances than just those where data compression is a main goal. This scheme involves using an index function for triangular arrays. This scheme is based on results given in a paper by David K. Chow (1969). It has been independently discovered by Richard A. Gustafson (1969, 1971).

It is particularly useful for key-values with only a few one bits occurring in general; thus we shall call this hashing function a sparse key hashing function. Such key-values exist when one codes binary attribute data or other binary coded data as a bit-string key-value, where only a few attributes apply for any one item (see (Shapiro *et al.*, 1969; Lum, 1970)).

The basic scheme is such that key-values having exactly s one-bits are spread in a balanced way among the addresses $0, 1, \ldots, n-1$, independently for every value of s.

Let our key-values be t-bit fields with the bit positions numbered $0, 1, \ldots, t-1$ from left to right. Now given a key-value x, we associate the sequence of integers $p_1, p_2, \ldots, p_s$ which are the s ordinal positions in x where a one-bit occurs. We have $0 \leqslant p_1 < p_2 < \ldots < p_s \leqslant t-1$, where, of course, s is also a function of x.

Now let f_s be a function of s arguments, $0 \leqslant p_1 < p_2 < \ldots < p_s \leqslant t-1$, which maps the $\binom{t}{s}$ vectors, $(p_1, p_2, \ldots, p_s)$, 1-1 onto the integers $0, 1, \ldots, \binom{t}{s} - 1$. f_s is a triangular index function. Then we may define our sparse key hashing function as $H(x) = f_s(p_1, p_2, \ldots, p_s) \bmod n$, where s and p_i, $1 \leqslant i \leqslant s$, are functions of x as defined above (another variant is $H(x) = \left\lfloor n f_s(p_1, \ldots, p_s) / \binom{t}{s} \right\rfloor$).

Note that key-values with the same number of one-bits are separated in so far as possible. Thus we have a cluster-destroying hashing function for clusters defined by the number of one-bits in a key-value.

Now we may specify $f_s(p_1, \ldots, p_s) = \sum_{1 \leqslant i \leqslant s} \binom{p_i}{i}$.

It is not hard to see that this is a 1-1 onto function. See Knuth (1968), Section 2.2.6.

We should observe that even with sparse keys the computation of $H(x)$ is time-consuming. A 'shift until a one and count' instruction, i.e. some quick way to obtain $p_1, p_2, \ldots, p_s$, as in floating-to-fixed conversion instructions may help.

Gustafson has noted that in the case where the key-value may be too long, one can prehash a key-value before using the sparse key hashing function, H. Compressing a key-value before using H also has the advantage that 'and-form retrievals with don't care conditions' can be more easily handled if the compression is properly done.

Thus it is particularly convenient in choosing a hashing function for such prehashing to insist that it preserve the following inclusion property. A bit string x will be said to be *included* in a bit string y if x and y are of the same length and whenever a bit of x is a 1 the corresponding bit of y is also 1. We write $x \subseteq y$. Note $x \subseteq y$ implies $x \vee y = y$ and $x \,\&\, y = x$.

Now we wish to choose a hashing function, Q, such that if $x \subseteq y$ then $Q(x) \subseteq Q(y)$. Such a function can be specified as follows:

First, decompose x into a sequence of subfields $x_1, \ldots, x_m$. Now permute the bits within each subfield as desired and then *or* and *and* these fields together in any way desired to obtain a final bit string x'. This process preserves inclusion and compresses x into a shorter bit string x' which may now be hashed with the sparse key hashing function H.

The benefit of choosing Q so as to preserve inclusion is that we can then process 'and-form' retrieval requests reasonably rapidly. If, for example, we wish to retrieve all items with key-value $x = 01X0X1$ where each X is a 'don't-care', then we merely search the buckets $H(Q(010001))$, $H(Q(010011))$, $H(Q(011001))$, and $H(Q(011011))$. In practice of course we may first compute $Q(x)$ in a formal manner, and then search the buckets specified by each possible specialisation of $Q(x)$.

Thus if $Q(01X0X1) = (01 \,\&\, X0) \vee X1$ then $Q(01X0X1) = X1$, whereupon we search the buckets $H(01)$ and $H(11)$. Moreover, we may drop all duplicate occurrences of the final bucket addresses so that each bucket is searched at most once.

Now the problem arises of how to choose Q so as to minimise collisions. Minimising collisions insures that we do not search too many buckets fruitlessly. In the language of superimposed coding (Stiassny, 1960), we wish to minimise the expected number of false drops. This can be done by judiciously choosing the subfields of x, the permutations of the bits thereof, and which initial and intermediate subfields are to be and-ed and which or-ed in the specification of Q. We shall discuss the problem of false drops in Section 3.2.

It should be pointed out that the less specific a retrieval request is, the more buckets must be searched. Thus for simple retrieval requests which specify the values of just a few bits, an inverted or multiply-indexed file organisation may be more appropriate. Superimposed hashing and multiple indexing are, in a sense, complementary schemes.

Other hashing functions of similar nature which require that multiple copies of some items be stored and which depend strongly upon particular parameters being satisfied have been studied. These schemes, which are based upon certain combinatorial relations which hold in finite geometries, may be found in the literature referenced by Chow (1969), and in Ghosh and Abraham (1968) and Ghosh (1968).

Wong and Chiang (1971) have noted that if we categorise the items being stored into sets, A_i, defined by $A_i = \{x|x_i = 1\}$ where each item has a t-bit key value $x_1:_t$, then the atoms (i.e. 'smallest sets') of the Boolean alegebra generated by $(A_1, A_2, \ldots, A_t)$ are just the sets of items with distinct key-values. By defining a hashing function which partitions the items into buckets which coincide with these atoms, one can subsequently build a very fast and general storage and retrieval system, where all Boolean retrieval requests are converted to disjunctive normal form wherein the atoms to be retrieved are explicitly shown. This latter device is a useful way of representing a retrieval request. As for obtaining a hashing function whose buckets coincide with the sets of distinct items, however, this is just the general storage and retrieval problem all over again, and the fact that these buckets are atoms of the Boolean algebra $(A_1, \ldots, A_t)$ is a useless fact.

Finally we should also point out that index functions are a special form of Gödel numbering (Hermes, 1965). A particular Gödel numbering scheme is to map the sequence of integers $i_1, i_2, \ldots, i_s$ into the value $p_{i_1}p_{i_2}, \ldots, p_{i_s}$ where p_j is the jth prime. Now if v is such a Gödel number, then i is one of the original indices if p_i divides v. If we use $v = p_1^{i_1}p_2^{i_2}, \ldots, p_s^{i_s}$ instead then we can determine if i is the jth index by testing whether p_j^i exactly divides v.

Although such an encoding based on products of primes or prime powers has attractive properties for handling and-form retrievals (Cockayne and Hyde, 1960), the values which result may be so large as to be impractical, and moreover the important divisibility properties are easily lost if any compression schemes are used. R. Creighton Buck (1961) has a scheme which achieves some slight compression, but his method is still far from being generally useful.

2.6. *Statistical behaviour of hashing functions*

It is of some interest to consider the expected behaviour of a balanced hashing function, H, with range $\{0, 1, \ldots, n-1\}$ under the assumption that the collection of items to be processed is a set of k items with key-values taken randomly without replacement from a set of m possible key-values.

Recall that we say synonymous items hash to the same bucket, and even though such items may not all be able to reside in that bucket we shall speak as though they do conceptually reside there, so that notions such as the number of items in excess of b in a given bucket (which we call the number of initial b-overflow items in that bucket) can make sense.

Let the random variable N_j be the number of items which hash to the jth bucket for $0 \leqslant j \leqslant n-1$. We have k items, so $N_0 + N_1 + \ldots + N_{n-1} = k$. Now under our assumptions, we have that each N_j is distributed according to the hypergeometric distribution (see Feller, 1957), so:

$$P(N_j = i) = \frac{\binom{\frac{m}{n}}{i}\binom{m - \frac{m}{n}}{k-i}}{\binom{m}{k}}$$

It is not convenient, or at any rate not traditional, to deal with expressions which involve the number of possible key-values, m. Rather we modify our assumptions to be that each of the k given items has probability $\frac{1}{n}$ of hashing to the jth bucket. This is equivalent to assuming we have a random sample of k items chosen with replacement. But we may not permit duplicate key-values, therefore it is more accurate to say we are assuming m is so large that we may approximate the hypergeometric distribution with the binomial distribution and thus avoid requiring knowledge of m.

Thus we have for any random key-value, X,

$$P(H(X) = i) = \frac{1}{n} \text{ for } 0 \leqslant i \leqslant n-1 . \tag{1}$$

From this we have:

$$p_i \overset{df}{=} P(N_j = i) = \binom{k}{i}\left(\frac{1}{n}\right)^i\left(1 - \frac{1}{n}\right)^{k-i} . \tag{2}$$

We immediately obtain the expected number of items per bucket:

$$E(N_j) = \frac{k}{n} . \tag{3}$$

Let S_i be the number of *i-full buckets*, that is the number of buckets which have exactly i items hashing to them. Then $S_i = \delta_{N_0 i} + \ldots + \delta_{N_{n-1} i}$ and:

$$E(S_{N_j i}) = P(N_j = i) \tag{4}$$

and so,

$$E(S_i) = \sum_{0 \leqslant j \leqslant n-1} E(\delta_{N_j i}) = \sum_{0 \leqslant j \leqslant n-1} p_i = np_i . \tag{5}$$

Now let T_b be the total number of *b-overflow items*, that is, T_b is the number of items which are preceded by b or more items in their respective buckets, for $0 \leqslant b$. Then T_b may be defined as:

$$T_b = \sum_{1 \leqslant i \leqslant k-b} i \cdot S_{b+i} . \tag{6}$$

We may think of T_b in figurative terms as the number of items at a 'level' greater than 'level' b.

Then from (6) and (5), we have for $0 \leqslant b$:

$$E(T_b) = \sum_{b+1 \leqslant j \leqslant k} (j-b)E(S_j) = n \sum_{b \leqslant j} (j-b)p_j . \tag{7}$$

In particular, the total number of initial collisions which can be expected is:

$$E(T_1) = k - n + n\left(1 - \frac{1}{n}\right)^k . \tag{8}$$

We can compute the variance of T_b as follows.

$$\text{Var}(T_b) = E((T_b - E(T_b))^2) = E(T_b^2) - E(T_b)^2 ,$$

and,

$$E(T_b^2) = \sum_{b \leqslant i,j} (i-b)(j-b)E(S_iS_j) ,$$

and,

$$E(S_iS_j) = \sum_{0 \leqslant s,t \leqslant n-1} E(\delta_{N_t i}\delta_{N_s j}) ,$$

and

$$E(\delta_{N_t i}\delta_{N_s j}) = P(N_t = i, N_s = j) =$$

$$\begin{cases} n^{1/k}\binom{k}{j}\binom{k-j}{i}(n-2)^{k-i-j} & \text{for } s \neq t \\ \delta_{ij} \cdot p_i & \text{for } s = t. \end{cases}$$

Then substituting back,

$$\mathrm{Var}\,(T_b) = n^2 \sum_{b \leqslant i,j} (i-b)(j-b)p_i p_j$$

$$\left[\frac{(k-j)^{\underline{i}}}{k^{\underline{i}}}\left(\frac{n}{n-1}\right)^{k-1}\left(\frac{n-2}{n-1}\right)^{k-j-1} - 1\right]$$

$$+\, n \sum_{b \leqslant i} (i-b)^2 p_i \,. \quad (9)$$

$E(T_b)$ has been computed by a different argument by John G. Williams (1971). His result yields

$$E(T_b) = n^{-b} \cdot \sum_{i \leqslant k} (k-i)\binom{i-1}{b-1}\left(1-\frac{1}{n}\right)^{i-b}.$$

It has often been noted that distributing items randomly among buckets does not result in a uniform distribution but rather in a binomial distribution which admits a Poisson approximation (Heising, 1958; Bucholz, 1963; Lin, 1963; Tainiter, 1963; Schay and Spruth, 1962).

Consider storing k items and then storing a random $(k+1)$st item. Let I be the number of items already in the bucket into which the $(k+1)$st item is stored. Then from (5),

$$P(I = i) = \frac{E(S_i)}{n} = p_i \,.$$

It is in this sense that the items are binomially distributed among the buckets, It is, of course, unreasonable to expect uniformity to result from a random assignment of items to buckets. Uniformity can sometimes be approached by distribution-dependent techniques as described in Sections 2.1 and 2.4, but in general we settle for hashing functions which satisfy (1).

One can compute a table of values, $E(S_i)$, for n and k fixed, using (5), which exhibits the expected distribution by showing the expected number of buckets which hold i items for $0 \leqslant i \leqslant k$.

For example, for $n = k = 10$, we have:

i	$E(S_i)$
0	3·49
1	3·87
2	1·94
3	0·57
4	0·11
5	0·01
6	0·001
7	0·0001
8	0·000001
9	0·0000001
10	0·000000001

3. A history of hashing schemes

The study and development of storage and retrieval algorithms is a recent phenomenon. Perhaps such issues were of no concern prior to the occurrence of the 'information explosion' and really massive bureaucracies. In any event it was only in the 1950's that the notion of hashing became current, although other storage and retrieval schemes were studied by archivists and librarians many years earlier (Mooers, 1959; Casey *et al.*, 1958).

3.1. *Content-addressable storage*

The general storage and retrieval problem can be solved by the use of a sufficiently general *content-addressable* storage device (Newell, 1961, Minker, 1971). In a famous article on future prospects, Vannevar Bush (1945) discussed a mythical system called 'memex' which might be considered analogous to an extremely flexible and powerful content-addressable store. Such a device would be able to hold variable length items and to present immediately upon demand a list of those items which shared a common value for any fixed but arbitrary field. Such a list could be further augmented or refined with the same mechanism. Unfortunately, current associative memories are much more restricted in their operation. Practical problems include fixed cell sizes, small capacities and difficulties with methods for dealing with the lists of items obtained.

Content-addressable memories often have an ordinary memory in parallel. The two together can be considered as a memory with words which have a content-addressable field and a non-content-addressable field. A particularly simple and versatile form of partially content-addressable memory is provided by edge-notched card systems where the edges form the content-addressable field. We shall consider edge-notched cards below. An early hardware development of this type is described in Hollander (1956).

One may generalise the notion of hashing to that of utilising such a partially content-addressable memory. The hashing function merely computes a hash value suitable for storage in the content-addressable field. Indeed hashing algorithms are often used to simulate limited content-addressable facilities (Newell, 1962; Feldman and Rovner, 1969). Thus we may consider hashing as a compression operation where we convert key-values into shorter values, and then file the items involved according to the shortened values, somehow resolving the problem of collisions. This paradigm was first applied in edge-notched card filing schemes by Mooers (1947) and Carl S. Wise (1947), although edge-notched and Hollerith cards were in existence long before. They simultaneously dealt with the more complex problem of the retrieval of items whose key-values match a partial key-value specification, i.e. and-form retrievals with don't-care conditions, although they probably did not think of their schemes in such terms. The process involved came to be known as superimposed coding (Wise, 1958) and it is still a viable technique (Hutton, 1968).

3.2. *Edge-notched cards and superimposed coding*

Edge-notched card filing systems are described by Robert S. Casey, James W. Perry, and others (1951; 1958). Also see Shapiro *et al.* (1969). Basically, we write the data and key-value of an item on a card (which may hold other such inscriptions) and then punch the binary-coded key-value into a predefined field of notchable positions along some edge of the card. Each position contains either a hole or a notch (which incorporates the hole). A zero is indicated by a hole while a one corresponds to a notch. One can retrieve all cards with a one or zero in a certain position by running a long needle through the holes and notches in that position of the cards. Upon lifting the needle, the notched cards stay behind while the unnotched cards remain on the needle. Thus the desired separation is effected.

There are elaborate schemes for coding data in binary without using an excessive number of one bits (when many positions are notched, correspondingly many strokes of the needle are required). Methods for sorting and for various forms of retrieval have been developed. Also, many variations of the basic idea have been developed. For example there are cards with two holes per position which permit two independent notches to be made in a position and thus allow radix 4 codes to be used at the price of some complicated needle operations to achieve separation.

We shall consider only 'binary' cards and we shall use the terms one and zero instead of notched and unnotched. Moreover we shall leave to the imagination the operations with needles which must be done to accomplish various discriminations which we whall describe functionally.

Superimposed coding consists of taking a key-value and or-ing segments of it together to obtain a shorter value. The item is entered upon a card and the compressed key-value is notched into an appropriate edge field of the card.

Obviously, one can retrieve all items with given completely or partially specified key-values by retrieving those cards with appropriately notched edge positions. Also, however, cards may be retrieved whose edge notches satisfy the retrieval criterion, but only because the encoded key-value collided with a desired key-value when it was or-ed by segments to compress it (i.e. when it was hashed). These unwanted cards which are retrieved due to collisions caused by superimposed coding are called *false drops*.

A good superimposed coding scheme must minimise the expected number of false drops. Of course the form of the retrievals to be done must be taken into account. The fundamental problem is balancing the use of an initially-chosen set of key-values which are relatively short, and hence need to be folded and or-ed only a few times, against the use of initial key-values which are longer and need more folding, but which are sparse in that there are only a few one bits per segment to be or-ed. Since, in practice, the number of needling operations is the important factor to be minimised rather than the size of the edge field the sparse key-value approach is most often used. Various superimposed coding schemes and the derivations of the associated expected number of false drops have been presented (Wise, 1958; Stiassny, 1960; Olson, 1969; Kautz and Singleton, 1964).

From the point of view of those who design various superimposed coding schemes, the method is a device (*a*) to minimise the number of cards required in the file, or (*b*) to minimise the number of edge positions required to code a key-value.

To achieve (*a*), several items are entered upon the same card, thus logically constructing a 'super-item'. The key-value of the super-item is taken to be the concatenation of the key-values of the various component items. The superimposition coding then consists of or-ing together the key-value segments corresponding to the key-values of the original component items.

Thus, we could equally well dispense with the notion of super-items and merely speak of entering several items in one card and *superimposing* their key-values in the common edge field. A retrieval request of course generally consists of specifying a key-value from the set of key-values of the original component items, or equivalently, of specifying an appropriate segment of a possible super-item key-value.

We may note at this point that it is generally desirable to reduce the number of cards in a file to the smallest number of 'needle-fulls' possible to avoid excess needling operations, even at the cost of somewhat more work per needle-full of cards and more false drops to be expected (see Shapiro *et al.*, 1969).

The retrieval and hash function selection considerations are much the same here as they are in the scheme of Gustafson and Chow presented in Section 2, with one exception. In an edge-notched file, a card corresponds to a item-storage cell (albeit one 'super-item') and the set of all cards with the same edge-field values correspond to a bucket of items. But unlike a computer-based scheme, a search through several buckets is not carried out sequentially by bucket and hence empty buckets generally cause no increase in cost. In the triangular index function method, we must probe to discover that a bucket is empty, but this is not the case in an edge-notched card file. If we are searching, for example, all buckets whose edge-notched key-value starts with 01, then if the bucket '0111' is empty, it does not have to be probed, it merely contributes no cards to the end result.

The use of superimposed coding to achieve (*b*) is even more in the mainstream of hash table methods. In this case one superimposes segments of the key-value to obtain a hashed value to be coded in the edge-notch field, and the same process is followed to obtain a value for retrieval purposes. The notion of a bucket is, just as above, the set of all cards with a common edge-notched value.

3.3. *Open addressing and chaining*

Hash table storage and retrieval was not immediately developed upon the advent of digital computers although the storage and retrieval problem was under attack from the beginning. This was the case even though the disguised use of hashing methods in the form of superimposed coding was well known by 1950 and some of the people who used edge-notched card files had by then joined the emerging programming profession. Probably the main difficulty was that, as we remarked above, needling corresponds to a parallel search, something that computers could not do, and hence, the possibility of collisions seemed to doom at the start any key-value-to-address scheme which was not 1-1.

In 1953, before the IBM 701 was formally marketed, Nathaniel Rochester, who had responsibility for 'software' for the new machine, was collaborating at Poughkeepsie with Arthur L. Samuel, Gene M. Amdahl, and Elaine M. Boehme who were building an assembler for the 701. Samuel reports (1970) that they decided to try a key-value-to-address mapping (probably either an extraction or division scheme) for managing the symbol table in their assembler. They at first used their hashing scheme without any method of resolving collisions. Then, spurred by necessity, Amdahl discovered linear open addressing.

Buchholz (1963) reports that Hans Peter Luhn and Andrew D. Lin were considering the problem of hashing and collisions at about the same time. They reported their results in some unpublished IBM reports dated February: March, 1953. They noted that they could partially solve the problem of collisions by providing buckets at each address, each of which could hold several items, thus reducing the probability of a fatal overflow. Such a solution can be considered to be a proto-chaining scheme. This scheme is mentioned in (Lesser and Haanstra, 1956). Moreover Luhn explicitly described direct chaining into a separate overflow area. He also proposed a primitive form of open addressing based on 'degenerative' addresses. The probe sequence for an item x with an n digit hash address is given by the successive k-digit prefixs of the hash value for $k = n - 1, n - 2, \ldots, 2, 1$. Other early efforts are described in Dumey (1965), EDP Analyser (1957), and Gottlieb (1960).

The widespread use of chaining methods had to await the popularisation of 'list-processing' provided by the work of Allen Newell, Herbert Simon, and J. Clifford Shaw (1957) on the Logic Theory machine and Newall and Tonge on IPL (1961).

In April 1957, W. Wesley Peterson published a detailed analysis of linear open addressing (1957) wherein the phrase 'open addressing' was coined. A note by Andrei P. Ershov in August 1958 indicates that he too was aware of linear open addressing and had made very ingenious use of the technique in a program for code optimisation in a compiler.

After Luhn and Lin (1953), the earliest published work on chaining as a method of collision resolution seems to be the description of the method as used with the IBM 305 RAMAC disc in 1958 (Heising, 1958). A paper by Francis A. Williams (1959) followed in June 1959 which describes direct chaining with coalescing lists.

A December 1961 memorandum by W. W. Peterson describes multiple hashing function collision resolution, whch is generally a split method, as reported to him by John Mauchly and states that the original inventor is unknown.

By the end of 1963 many different schemes had been developed (Schay and Raver, 1963; McIlroy, 1963; Johnson, 1961;

Schay and Spruth, 1962) and hashing algorithms were firmly established as an important item in the programmer's repertoire). Since then several useful surveys have been published, notably Buchholz (1963) and Morris (1968).

D. E. Knuth's recent work, *The Art of Computer Programming: Vol.* 3, *Sorting and Searching* (1973) is the next milestone in the history of this method. Most of the topics covered here, as well as a number of other points, notably the resolution of many heretofore unanalysed issues and algorithms are treated by Knuth. The subject is not completely closed, however, and many problems still await solution.

In recent years the more general notion of hashing as a compression transformation independent of an address computation has become popular. People often speak of hashing methods, meaning a scheme for compression, without regard to storage and retrieval issues at all just as in early applications with edge-notched cards. Martin (1964), for example, has developed a 'quick' partial test for the identity of two functions based on testing the equality of the functions evaluated on several random arguments. He calls these values, hash codes. Another use of 'hash-coding' in this generalised sense is in Floyd's algorithm for testing for isomorphic graphs (1969). Thus the concept of hashing finds wider application than just in computing addresses. It is a basic concept which can be useful in many circumstances.

Appendix 1 d-separating hashing functions and the BCH theorem

Recall that given a finite field, $GF(q)$, of q elements, there exist extension fields $GF(q^m)$ for every m. $GF(q^m)$ can be represented as the set of polynomials of degree at most $m-1$ with coefficients in $GF(q)$. Addition is taken as ordinary polynomial addition, while multiplication is defined as polynomial multiplication modulo an arbitrary but fixed polynomial, $f(z)$, of degree m which is irreducible over $GF(q)$. We write

$$GF(q^m) = GF(q)[z]/(f(z))$$

where $GF(q)[z]$ is the set of all polynomials in z over $GF(q)$, $(f(z))$ is the principal ideal in $GF(q)[z]$ consisting of all multiples of $f(z)$, and $GF(q)[z]/(f(z))$ is the set of residue classes induced by the homomorphic mapping which carries the members of $(f(z))$ to 0. Thus, to be precise $GF(q^m)$ is a set of residue classes; but in practice we often choose representatives of these classes, whence we consider $GF(q^m)$ as the set of polynomials of degree at most $m-1$ in $GF(q)[x]$.

$GF(q)$ is embedded in $GF(q^m)$ in the obvious way as the subset of constant polynomials. $f(z)$ as a polynomial over $GF(q^m)$ splits or factors into linear factors with coefficients in $GF(q^m)$, hence $GF(q^m)$ is called the splitting field of $f(z)$.

$GF(q)$ is essentially unique in that any two q-element fields are isomorphic, however, there are many different representations for $GF(q)$.

We also recall that $GF(q) - \{0\}$ is isomorphic as a multiplicative group to the $q-1$ qth roots of unity. Accordingly, $GF(q) - \{0\}$ is a cyclic group. An element of $GF(q)$ which generates this group is called a *primitive element* of $GF(q)$. We define $o(a)$ as the least integer n such that $a^n = 1$ in $GF(q)$. If a is a primitive element of $GF(q)$ then $o(a) = q - 1$.

A primitive element of $GF(q^m)$ is a root of a unique irreducible polynomial of degree m in $GF(q)[z]$. In fact, for any element $a \in GF(q^m)$ there is a unique irreducible monic polynomial, $m_a(z)$ in $GF(q)[z]$ which has a as a root. $m_a(z)$ is called the *minimal polynomial of a over* $GF(q)$. $m_a(z)$ is not independent of the representation of $GF(q^m)$ as an algebra over $GF(q)$. The degree of $m_a(z)$ however is invariant. In any event $m_a(z)$ divides $z^{q^m} - z$ for every $a \in GF(q^m)$ and hence $lcm\{m_a(z) | a \in GF(q^m)\}$ divides $z^{q^m} - z$.

For a systematic introduction to the theory of finite fields, see van der Waerden (1931) or Peterson (1961). The above review is sufficient for our purposes.

The following theorem of Bose, Chauduri and Hocquenghem (1960), (Peterson, 1961), tells us how to construct polynomials to use in d-separating hashing functions.

Theorem (*Bose, Chauduri, Hocquenghem*):
Let $a \in GF(q^m)$ for some m, be such that $o(a) \geqslant t$. Let j and d be integers and let $g(z)$ be the polynomial of degree s over $GF(q)$ defined by $g(z) = lcm\,(m_{a^j}(z), m_{a^{j+1}}(z), \ldots, m_{a^{j+d-1}}(z))$ where $m_\alpha(z)$ is the minimum polynomial of α over $GF(q)$. Thus, $a^j, a^{j+1}, \ldots, a^{j+d-1}$ are roots of $g(z)$. j is arbitrary but $d \leqslant s \leqslant o(a)$. Let x and y be distinct t-tuples in $K(T_g)$, then $D(x, y) > d$, where D is the Hamming metric. Thus $K(T_g)$ has minimum distance $\geqslant d + 1$.

Proof:
Recall that $K(T_g)$ is the kernel of T_g where T_g is the $s \times t$ matrix such that $T_g x = (x(z) \bmod g(z))^*$. We must show that if x and y are distinct t-component column vectors over $GF(q)$ such that $T_g x = T_g y = 0$, then $D(x, y) > d$.

But $T_g x = 0$ iff $x(z) \bmod g(z) = 0$ whence $g(z)$ divides $x(z)$, and thus if $T_g x = 0$ then $x(z)$ has the values $a^j, a^{j+1}, \ldots, a^{j+d-1}$ (which lie among the roots of g), as roots. This can be restated as $T_g x = 0$ where $T_g = A$ and

$$A_{s \times t} = \begin{bmatrix} 1 & a^j & (a^j)^2 & \ldots & (a^j)^{t-1} \\ 1 & a^{j+1} & (a^{j+1})^2 & \ldots & (a^{j+1})^{t-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & a^{j+d-1} & & \ldots & (a^{j+d-1})^{t-1} \end{bmatrix},$$

since

$$Ax = \begin{bmatrix} x(a^j) \\ x(a^{j+1}) \\ \vdots \\ x(a^{j+d-1}) \end{bmatrix}.$$

Thus, we may assume $Ax = Ay = 0$ and we must show $D(x, y) > d$.

Now we shall show that every set of d columns of A are linearly independent over $GF(q^m)$. From this we may use Sack's theorem given earlier to deduce that since $Ax = Ay$ while $x \neq y$ we must have $D(x, y) > d$ as desired.

Consider the determinant of the matrix formed from an arbitrary set of d distinct columns from A.

$$\det \begin{bmatrix} (a^j)^{i_1} & \ldots & (a^j)^{i_d} \\ \vdots & & \\ (a^{j+d-1})^{i_1} & \ldots & (a^{j+d-1})^{i_d} \end{bmatrix} =$$

$$a^{j(i_1 + i_2 \ldots + i_d)} \cdot \det \begin{bmatrix} 1 & 1 & \ldots & \\ a^{i_1} & a^{i_2} & \ldots & \\ \vdots & \vdots & & \\ (a^{i_1})^{d-1} & & \ldots & (a^{i_d})^{d-1} \end{bmatrix} =$$

$$a^{j(i_k + \ldots i_d)} \cdot \prod_{1 \leqslant \alpha < \beta \leqslant d} (a^i\beta - a^i\alpha)$$

since we have obtained a Van der Monde determinant. Now $a^i\beta \neq a^i\alpha$ for $\beta \neq \alpha$. This follows since $i_\beta, i_\alpha < t \leqslant o(a)$. Hence we have a product of non-zero field elements as the value of det A and so det $A \neq 0$.

But det $A \neq 0$ implies that our set of d columns is linearly independent over $GF(q^m)$ and since this set was arbitrary, we have that every set of d columns of A is linearly independent and thus we are done. Q.E.D.

The set $K(T_g)$ in the BCH theorem is known as a BCH cyclic $(t, t-s)$-code with minimum distance $\geqslant d + 1$.

Now by our previous comments on choosing $g(z)$ so that $H(x) = (x(z) \bmod g(z))^*$ is d-separating, we see that choosing $a \in GF(q^m)$ of order t and then choosing $g(z)$ of degree s with coefficients in the symbol field such that $g(z)$ has $a^j, a^{j+1}, \ldots,$

a^{j+d-1} as roots gives us a d-separating hashing function which maps t-tuples to s-tuples. Of course, there may be no such polynomials, $g(z)$, for certain t, s, and d, but for t and d fixed the BCH theorem gives us a construction for $g(z)$ whose degree then determines s.

One class of polynomials (which are called generators of Reed-Solomon codes) is obtained by taking a to be a primitive element of $GF(q)$, the same field which is to hold the coefficients of $g(z)$. Thus we take $m = 1$, and of course $t \leqslant o(a) = q - 1$. Now let

$$g(z) = \prod_{1 \leqslant i \leqslant s} (x - a^i) \ .$$

Then by the BCH theorem $K(T_g)$ has minimum distance $\geqslant s + 1$ (in fact in this case exactly $s + 1$) and hence by our previous remarks, $H(x) = (x(z) \bmod g(z))^*$ which maps t-tuples over $GF(q)$ to s-tuples over $GF(q)$, is an s-separating hashing function.

As another example, let $t = 15$ and choose a as a primitive element of $GF(2^4)$ so that $o(a) = t$. Take $j = 1$ and $d = 2$ in the BCH theorem, so that $g(z) = lcm(m_a(z), m_a2(z))$.

Now, $GF(2^4)$ may be defined as the residue classes of polynomials over $GF(2)$ modulo $f(z) = z^4 + z + 1$, with a such that $f(a) = 0$ (i.e. $a^4 + a + 1 \equiv 0 \bmod f(z)$). Taking this representation we get $m_a(z) = f(z)$.

Now since $f(a^p) = f(a)^p$ in a field of characteristic p we have that a^2 as well as a is a root of $f(z)$. But $o(a) = 15$. Hence a^2 is a primitive element of $GF(2^4)$ and so the degree of $m_{a^2}(z)$ is 4. But then $m_{a^2}(z) = f(z)$. Thus, $g(z) = 1 + z + z^4$, and $H(x) = (x(z) \bmod g(z))^*$ maps 15-bit strings to 4-bit strings and is 2-separating.

More elaborate results of coding theory than the BCH theorem are applicable to finding d-separating hashing functions. The fundamental result is that the process of computing the syndrome for a linear code of minimum distance $d + 1$ is a d-separating hashing function.

References

ACKERMAN, F. (1974). Quadratic Search for Hash Tables of Size p^n, *CACM*, Vol. 17, No. 3, p. 164.

AMBLE, O., and KNUTH, D. E. (1973). Ordered Hash Tables, *The Computer Journal*, Vol. 17, No. 2,pp. 135-142, May 1974; also Computer Science Department Report STAN-CS-73-367, Stanford University.

BARTON, I., CREASEY, S. E., LYNCH, M. F., SNELL, M. J. (1974). An Information-Theoretic Approach to Text Searching in Direct Access Systems, *CACM*, Vol. 17, No. 6, pp. 345-350.

BATAGELJ, V. (1975). The Quadratic Hash Method when the Table Size is not a Prime Number, *CACM*, Vol. 18, No. 4, pp. 216-217.

BATSON, A. (1965). The Organization of Symbol Tables, *CACM*, Vol. 8, No. 2, pp. 111-112.

BAYS, C. (1973). The Reallocation of Hash-Coded Tables, *CACM*, Vol. 16, No. 1, pp. 11-14.

BAYS, C. (1973). A Note on when to Chain Overflow Items within a Direct-Access Table, *CACM*, Vol. 16, No. 1, pp. 46-47.

BAYS, C. (1973). Some Techniques for Structuring Chained Hash Tables, *The Computer Journal*, Vol. 16, No. 2, pp. 126-131.

BELL, J. R. (1970). The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering, *CACM*, Vol. 13, No. 2, pp. 103-105.

BELL, J. R., and KAMAN, C. H. (1970). The Linear Quotient Hash Code, *CACM*, Vol. 13, No. 11, pp. 675-677.

BEYER, J. D., and MAURER, W. D. (1968). Letter to the editor, *CACM*, Vol. 11, No. 5, p. 378.

BJORK, H. (1971). A Bi-unique Transformation into Integers of Identifiers and Other Variable-Length Items, *BIT*, Vol. 11, No. 1, pp. 16-20.

BLOOM, B. H. (1970). Space/Time Trade-offs in Hash Coding with Allowable Errors, *CACM*, Vol. 13, No. 7, pp. 422-426.

BOOKSTEIN, A. (1972). Double Hashing, *J. Amer. Soc. Info. Sci.*, Vol. 23, No. 6, pp. 402-405.

BOOKSTEIN, A. (1973). On Harrison's Substring Testing Technique, *CACM*, Vol. 16, No. 3, pp. 180-181.

BOSE, R. C., and RAY-CHAUDHURI, D. K. (1960). On a Class of Error-Correcting Binary Group Codes, *Inf. and Control*, Vol. 3, No. 1, pp. 68-79.

BRENT, R. P. (1973). Reducing the Retrieval Time of Scatter Storage Techniques, *CACM*, Vol. 16. No. 2, pp. 105-109.

BUCHHOLZ, W. (1963). File Organization and Addressing, *IBM Systems Journal*, Vol. 2, pp. 86-111.

BUCK, R. C. (1961). Studies in Information Storage and Retrieval on the Use of Godel Indices in Coding, *Amer. Doc*, Vol. 12, No. 3, pp. 165-171.

BUSH, V. (1945). As We May Think, *Atlantic Monthly*, Vol. 176, No. 1, pp. 101-108; also reprinted in *The Growth of Knowledge*, edited by Manfred Kochen, Wiley, N.Y., 1967.

BEATTY, J., and MUROGA, S. (1965). File Memory Addressing, in *Some Problems in Information Science*, edited by Manfred Kochen, pp. 206-216, Scarecrow Press, N.Y., 1965.

CASEY, R. S., and PERRY, J. W. (eds.) (1951). *Punched Cards*, Reinhold Publishing Co., N.Y., 1951.

CASEY, R. S., PERRY, J. W., BERRY, M. M., and KENT, A. (eds.) (1958). *Punched Cards*, 2nd edition, Reinhold Publishing Co., N.Y., 1958.

CHIEN, R. T., and FRAZER, D. (1966). An application of Coding Theory to Document Retrieval, *IEEE Transactions on Information Theory*, Vol. IT-12, No. 2, pp. 92-96.

CHOW, D. K. (1900). *A Geometric Approach to Coding Theorem with Application to Information Retrieval*, Report R-368 (Dissertation), University of Illinois, Urbana, Illinois.

CHOW, D. K. (1969). New Balanced-File Organization Schemes, *Inf. and Control*, Vol. 15, No. 5, pp. 377-396.

COCKAYNE, A. H., and HYDE, E. (1960). Prime Number Coding for Information Retrieval, *The Computer Journal*, Vol. 3, No. 0, pp. 21-22.

COFFMAN, E. G. Jr., and EVE, J. (1970). File Structures Using Hashing Functions, *CACM*, Vol. 13, No. 7, pp. 427-432, 436.

DAY, A. C. (1970). Full Table Quadratic Searching for Scatter Storage, *CACM*, Vol. 13, No. 8, pp. 481-482.

DEBALBINE, G. (1967). Thesis, California Institute of Technology.

DE LA BRIANDAIS, R. (1959). File Searching Using Variable Length Keys, *WJCC Proc.*, pp. 295-298.

DUMEY, A. I. (1965). Considerations on Random and Sequential Arrangements of Large Numbers of Records, *Proc. IFIP Congress*, Vol. 1, pp. 255-260.

DUMEY, A. I. (1956). Indexing for Rapid Random-Access Memory, *Computers and Automation*, Vol. 5, No. 12, pp. 6-9.

ECKER, A. (1974). The Period of Search for the Quadratic and Related Hash Methods, *The Computer Journal*, Vol. 17, No. 4, pp. 340-343.

ELCOCK, E. W. (1965). Note on the Addressing of Lists by Theire Source-Language Names, *The Computer Journal*, Vol. 8, No. 3, pp. 252-243.

ERSHOV, A. P. (1958). On Programming Arithmetic Operations, *CACM*, Vol. 1, No. 8, pp. 3-6.

FELDMAN, J. A., and LOW, J. R. (1973). Comment on Brent's Scatter Storage Algorithm, *CACM*, Vol. 16, No. 11, p. 703.

FELDMAN, J. A., and ROVNER, P. D. (1969). An Algol-Based Associative Language, *CACM*, Vol. 12, No. 8, pp. 439-449.

FELLER, W. *An Introduction to Probability Theory and its Applications*, Vol. 1, second edition, Wiley and Sons Inc., 1957.

FILES, J. R., HUSKEY, H. D. (1969). An Information Retrieval System Based on Superimposed Coding, *Proc. AFIPS* 1969, FJCC, Vol. 35, pp. 423-432, AFIPS Press, Montvale, N.J.

FLOYD, R. W. (1969). An Algorithm for Testing Graph Isomorphism, unpublished, Stanford University.

FLOYD, R. W. (1970). Personal communication.

FORBES, K. (1972). Random Files and Subroutine for Creating a Random Address, *Australian Computer Journal*, Vol. 4, No. 1, pp. 35-40.

FREIMAN, C. V., and CHIEN, R. T. (1963). Further Results in Polynomial Addressing, *IBM Journal of Research and Development*, Vol. 7, No. 4, October 1963 (letter to the editor).
GHOSH, S. P., and ABRAHAM, C. (1968). Application of Finite Geometry in File Organization for Records with Multiple-Valued Attributes, *IBM Journal of Research and Development*, Vol. 12, No. 2, pp. 180-187.
GHOSH, S. P. (1969). On the Problem of Query Qriented Filing Schemes Using Discrete Mathematics, *1968 IFIP Proc.*, Vol. II, ed. A. J. H. Morrell, pp. 1226-1232, North-Holland Publishing Co., Amsterdam.
GHOSH, S. P., and SENKO, M. E. (1969). File Organization: On the Selection of Random Access Index Points for Sequential Files, *JACM*, Vol. 16, No. 4, pp. 569-579.
GOTLIEB, C. C. (1960). General-Purpose Programming for Business Applications, in *Advances in Computers*, Vol. 1, pp. 1-42, ed. L. A. Franz, Academic Press, N.Y.
GURSKI, A. (1973). A Note on Analysis of Keys for use in Hashing, *BIT*, Vol. 13, No. 1, pp. 120-122.
GUSTAFSON, R. A. (1969). *A Randomized Combinational File Structure for Storage and Retrieval Systems*, Thesis, University of South Carolina, Columbia, S.C.
GUSTAFSON, R. A. (1971). Elements of the Randomized Combinatorial File Structure, Proc. of the Symposium on Inf. Storage and Retrieval, *ACM SIGIR*, pp. 163-174, University of Md.
HALTON, J. H. (1965). The Distribution of the Sequence $\{n\zeta\}$ $(n = 0, 1, 2, \ldots,)$, *Cambridge Philosophical Society Proceedings*, Vol. 61, pp. 665-670.
HAMMING, R. W. (1950). Error-Detecting and Error Correcting Codes, *BSTJ*, Vol. 26, No. 2, pp. 147-160.
HANAN, M., and PALERMO, F. P. (1963). An Application of Coding Theory to a File Address Problem, *IBM Journal of Research and Development*, Vol. 7, No. 2, pp. 127-129.
HARRISON, M. C. (1971). Implementation of the Substring Test by Hashing, *CACM*, Vol. 14, No. 12, pp. 777-779
HEISING, W. P. (1958). Methods of File Organization for Efficient Use of IBM RAMAC Files, *Proc. of the WJCC*, pp. 194-196, AIEE.
HEISING, W. P. (1963). File Organization and Addressing, *IBM Systems Journal*, Vol. 2, pp. 86-111.
HELLERMAN, H. (1967). *Digital Computer System Principles*, sections 3.14-3.17, McGraw-Hill, New York, N.Y.
HERMES, H. (1965). *Enumerability-Decidability-Computability*, Academic Press, N.Y.
HIGGINS, L. D., and SMITH, F. J. (1971). Disc Access Algorithms, *The Computer Journal*, Vol. 14, No. 3, pp. 249-253.
HOCQUENGHEM, A. (1959). Codes Correcteurs D'erreurs, *Chiffres*, Vol. 2, pp. 147-156.
HOLLANDER, G. L. (1956). Quasi-Random Access Memory Systems, *Proc. of the EJCC*, pp. 128-135, AIEE.
HOPGOOD, F. R. A., and DAVENPORT, J. (1972). The Quadratic Hash Method when the Table Size is a Power of 2, *The Computer Journal*, Vol. 15, No. 4, pp. 314-315.
HOPGOOD, F. R. A. (1968). A Solution for the Table Overflow Problem for Hash Tables, *The Computer Bulletin*, Vol. 11, No. 4, pp. 297-300.
HOPGOOD, F. R. A. (1969). *Compiling Techniques*, American Elsevier Inc., New York, N.Y.
HUTTON, F. C. (1968). PEEKABIT, Computer Offstring of Punched Card PEEKABOO, for Natural Language Searching, *CACM*, Vol. 11, No. 9, pp. 595-598.
ISAAC, E. J., and SINGLETON, R. C. (1956). Sorting by Address Calculation, *JACM*, Vol. 3, No. 2, pp. 169-174.
JOHNSON, L. R. (1961). An Indirect Chaining Method for Addressing on Secondary Keys, *CACM*, Vol. 4, No. 5, pp. 218-222.
KAUTZ, W. H., and SINGLETON, R. C. (1964). Nonrandom Binary Superimposed Codes, *IEEE Transactions on Information Theory*, Vol. IT-10, No. 4, pp. 363-377.
KLINKHAMER, J. F. (0000). *On Key-to-Address Transformation for Mass Storage*, University of Utah, Computer Science Department Report.
KNOTT, G. D. (1971). Expandable Open Addressing Hash Table Storage and Retrieval, *Proc. of the SIGFIDET Workshop on Data Description, Access and Control*, pp. 186-206, ACM.
KNUTH, D. E. (1968). *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, Addison Wesley, Reading, Mass.
KNUTH, D. E. (1969). *The Art of Computer Programming*, Vol. 2: *Semi-numerical Algorithms*, Addison Wesley, Reading, Mass.
KNUTH, D. E. (1973). *The Art of Computer Programming*, Vol. 3: *Searching and Sorting*, Addison-Wesley, Reading, Mass.
KNUTH, D. E. (1974). Computer Science and its Relation to Mathematics, *Amer. Math. Monthly*, Vol. 81, No. 4, pp. 323-343.
KONHEIM, A. G., and WEISS, B. (1966). An Occupancy Discipline and Applications, *SIAM Journal of Applied Mathematics*, Vol. 14, No. 6, pp. 1266-1274.
KOROLEV, L. N. (1958). Coding and Code Compression, *JACM*, Vol. 5, No. 4, pp. 328-330.
KRAL, J. (1971). Some Properties of the Scatter Storage Technique with Linear Probing, *The Computer Journal*, Vol. 14, No. 2, pp. 145-149.
KRONMAL, R. A., and TARTER, M. E. (1965). Cumulative Polygon Address Calculation Sorting, *Proc. of the 20th Natl. ACM Conf.*, pp. 376-384.
LAMPORT, L. (1970). Comment on Bell's Quadratic Quotient Method for Hash Code Searching, *CACM*, Vol. 13, No. 9, pp. 573-574.
LESSER, M. L., and HAANSTRA, J. W. (1956). The RAMAC Data Processing Machine: System Organization of the IBM 305, *Proc. of the EJCC*, pp. 139-146, AIEE. (See especially the discussion following on p. 146 where hashing to buckets is mentioned.
LIN, A. D. (1963). Key Addressing of Random Access Memories by Radix Transformation, *AFIPS SJCC Proc.*, Vol. 23, pp. 355-366.
LUCCIO, F. (1972). Weighted Increment Linear Search for Scatter Tables, *CACM*, Vol. 15, No. 12, pp. 1045-1047.
LUHN, H. P., and LIN, A. D. (1953). *Address System for Random Access Storage* (Code 01.011.450) and *Some Design Criteria for an Address System for Random Access Storage*, unpublished, IBM.
LUM, V. Y., and YUEN, P. S. T. (1972). Additional Results on Key-to-Address Transform Techniques: A fundamental Performance on Large Existing Formatted Files, *CACM*, Vol. 15, No. 11, pp. 996-997.
LUM, V. Y. (1973). General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept, *CACM*, Vol. 16, No. 10, pp. 603-612.
LUM, V. Y., and SENKO, M. E. (1973). *On the Equivalence of Overflow Handling Algorithms in Hash Coding*, IBM Research Report RJ-1279) (No. 20148), Watson Research Center.
LUM, Y. V. (1970). Multi-Attribute Retrieval with Combined Indexes, *CACM*, Vol. 13, No. 11, pp. 660-665.
LUM, Y. V., YUEN, P. S. T., and DODD, M. (1971). Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files, *CACM*, Vol. 14, No. 4, pp. 228-239.
MARTIN, W. A. (1964). *Hash-Coding Functions for a Complex Variable*, MIT AI Project Memo 70, MAC-M-165.
MAURER, W. D. (1968). An Improved Hash Code for Scatter Storage, *CACM*, Vol. 11, No. 1, pp. 35-38.
MAURER, W. D., and LEWIS, T. G. (1975). Hash Table Methods, *Computing Surveys*, Vol. 7, No. 1, pp. 5-20.
MCILROY, M. D. (1963). A Variant Method for File Searching, *CACM*, Vol. 6, No. 1, pp. 101.
MEALY, G. H. (1967). A Generalized Assembly System (Excerpts), in *Programming Systems and Languages*, Saul Rosen (ed.) pp. 535-571, McGraw-Hill.
MINKER, J. (1971). *An Overview of Associative or Content-Addressable Memory Systems and a KWIC Index to the Literature*, University of Maryland Comp. Sci. Tech. Report No. TR-157, University of Md.

MINSKY, M., and PAPERT, S. (1971). On Some Associative, Parallel, and Analog Computations, pp. 27-50 in *Associative Information Techniques*, Edwin L. Jacks (ed.), American Elsevier.

MOOERS, C. N. (1947). Putting Probability to Work, *Am. Chem. Soc. Abst.*, 14-15E, presented at the 112th National Meeting of the Am. Chem. Soc., New York.

MOOERS, C. N. (1959). The Next Twenty Years in Information Retrieval: Some Goals and Predictions, *Proc. WJCC*, pp. 81-86, IRE.

MORRIS, R. (1968). Scatter Storage Techniques, *CACM*, Vol. 11, No. 1, pp. 38-44.

MULLEN, K. A. (1971). Hash Coding and Optimally Ordered Entries, *Project THEMIS Tech. Report*, No. 42, Iowa University.

MULLIN, J. K. (1972). An Improved Index Sequential Access Method Using Hashed Overflow, *CACM*, Vol. 15, No. 5, pp. 301-307.

MUROGA, S. (1961). Unpublished report, IBM.

NEWELL, A., and SHAW, J. C. (1957). Programming the Logic Theory Machine, *Proc. WJCC*, pp. 218-240.

NEWELL, A. (1961). *A Note on the Use of Scrambled Addressing for Associative Memories*, unpublished, revised December 1962.

NEWELL, A., TONGE, F. M., FEIGENBAUM, E. A., GREEN, B. F. Jr., and MEALLY, G. H. (1964). *Information Processing Language-V Manual*, The Rand Corporation, 1961, second edition published by Prentice-Hall, 1964.

OLSON, C. A. (1969). Random Access File Organization for Indirectly Addressed Records, *Proc. ACM 24th National Conference*, pp. 539-549, ACM, N.Y.

OROSZ, G., and TAKACS, L. (1956). Some Probability Problems Concerning the Marking of Codes into the Superimposition Field, *The J. of Doc.*, Vol. 12, No. 4, pp. 231-234.

PETERSON, W. W. (1957). Addressing for Random Access Storage, *IBM Journal of Research and Development*, Vol. 1, No. 2, pp. 130-146.

PETERSON, W. W. (1961). *Error-Correcting Codes*, MIT Press.

PETERSON, W. W. (1961). Unpublished memorandum.

PRICE, C. E. (1971). Table Lookup Techniques, *Computing Surveys*, Vol. 3, No. 2, pp. 49-65.

RADKE, C. E. (1970). The Use of Quadratic Residue Research, *CACM*, Vol. 13, No. 2, pp. 103-105.

ROBERTS, D. C. (1972). File Organization Techniques, in *Advances in Computers*, edited by Morris Rubinoff, Vol. 12, pp. 115-174, Academic Press, N.Y.

ROTHNIE, J. B., Jr., and LOZANO, T. (1974). Attribute Based File Organization in a Page Memory Environment, *CACM*, Vol. 17, No. 2, pp. 63-69.

SACKS, G. E. (1958). Multiple Error Correction by Means of Parity Checks, *IRE Trans.*, Vol. IT-4, pp. 145-147.

SAMUEL, A. L. (1970). Personal communication.

SCHAY, G. Jr., and SPRUTH, W. G. (1962). Analysis of a File Addressing Method, *CACM*, Vol. 5, No. 8, pp. 459-462.

SCHAY, G. Jr., and RAVER, N. (1963). A Method for Key-to-Address Transformation, *IBM Journal of Research and Development*, Vol. 7, No. 2, pp. 121-126.

SCHUYLER, S. T., SOULE, J. A., TURCHETTA, P. F., and GOTTERER, M. H. (1972). An Image Matrix for Processing Files, *Proc. ACM 1972 Annual Conf.*, pp. 482-492.

SEVERANCE, D. G. (1974). Identifier Search Mechanisms: A Survey and Generalized Model, *Computing Surveys*, Vol. 6, No. 3, pp. 175-194.

SHAPIRO, R. M., SAINT, H., MILLSTEIN, R. E., HOLT, A. W., WARSHALL, S., and SEMPLINER, L. (1969). *A Handbook on File Structuring*, ADR Inc. Report RADC-TR-69-313, Vol. 1, Federal Clearinghouse No. AD697025.

SIMON, J. C., and GUIHO, G. (1972). On Algorithms Preserving Neighborhood, to File and Retrieve Information in a Memory, *Int. J. Computer and Inf. Sci.*, Vol. 1, No. 1, pp. 3-15.

SHNEIDERMAN, B. (1973). Polynomial Search, *Software Practice and Experience*, Vol. 3, No. 2, pp. 5-8.

STIASSNY, S. (1960). Mathematical Analysis of Various Superimposed Coding Methods, *Amer. Doc.*, Vol. 11, No. 2, pp. 155-169.

TAINITER, M. (1963). Addressing for Random-Access Storage with Multiple Bucket Capacities, *JACM*, Vol. 10, No. 3, pp. 307-315.

TARTER, M. E., and KRONMAL, R. A. (1968). Estimation of the Cumulative by Fourier Series Methods and Application to the Insertion Problem, *Proc. of the 23rd National ACM Conf.*, pp. 491-497, ACM.

TOYODA, J., TEZUKA, Y., and KASHARA, Y. (1966). Analysis of the Address Assignment Problem for Clustered Keys, *JACM*, Vol. 13, No. 4, pp. 526-532.

ULLMAN, J. D. (1970). A Note on the Efficiency of Hashing Functions, *JACM*, Vol. 19, No. 3, pp. 569-575, July 1972, also available as *The Design of Hashing Functions*, EE Dept. Tech. Report No. 85, Princeton University. This work is also discussed in Knuth, D. E. (1973).

VAN DER POOL, J. A. (1973). Optimum Storage Allocation for A File in Steady State, *IBM Journal of Research and Development*, pp. 27-38.

VAN DER POOL, J. A. (1973-1974). Optimum Storage Allocation for a File with Open Addressing, *IBM J. of Research and Development*, pp. 106-114.

VAN DER WAERDEN, B. L. (1931). *Modern Algebra*, Frederick Ungar Pub. Co., N.Y., 1940 (original German edition 1931).

WEBB, D. A. (1972). The Development and Application of an Evaluation Model for Hash Coding Systems, Ph.D. thesis, Syracuse Univ. (See *Computing Reviews*, Feb. 1973, review no. 24509).

WEISS, E. (1962). Compression and Coding, *IRE Transactions on Information Theory*, Vol. IT-8, No. 3, pp. 256-257.

WELSH, T. A. (1971). Bounds on Information Retrieval Efficiency in Static File Structures, *MIT Project MAC*, Thesis.

WILLIAMS, F. A. (1959). Handling Identifiers as Internal Symbols in Language Processors, *CACM*, Vol. 2, No. 6, pp. 21-24.

WILLIAMS, J. G. (1971). Storage Utilization in a Memory Hierarchy when Storage Assignment is Performed by a Hashing Algorithm, *CACM*, Vol. 14, No. 3, pp. 172-175.

WISE, C. S. (1947). Generalized Punched-Card Codes for Chemical Bibliographies, *Am. Chem. Soc. Abst.* 16E, presented at the 112th National Meeting of the Am. Chem. Soc. in New York, N.Y. September 1947. See also Chapter 6 in Casey, R. S., and Perry, J. W. (1951).

WISE, C. S. (1958). Mathematical Analysis of Coding Systems, Chapter 20, in Casey, R. S., Perry, J. W., Berry, Madeline M., and Kent, A. (1958). *Punched Cards.*

WONG, E., and CHIANG, T. C. (1971). Canonical Structure in Attribute Based File Organization, *CACM*, Vol. 14, No. 9, pp. 593-597.

Data Management: File Organization, *EDP Analyzer*, December 1957.

ZAREMBA, S. K. (1966). Good Lattic Points, Discrepancy, and Numberical Integration, *Annali de Matematica Pura ed Applicata*, Series 4 Vol. 73, pp. 293-317.